On the Feasibility of a Unified Modelling and Programming Paradigm

Haxthausen, Anne Elisabeth; Peleska, Jan

Published in:
Proceedings of the 7th International Symposium on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA 2016)

Link to article, DOI:
10.1007/978-3-319-47169-3_4

Publication date:
2016

Document Version
Peer reviewed version

On the Feasibility of a Unified Modelling and Programming Paradigm

Anne E. Haxthausen[1] and Jan Peleska[2]

[1] DTU Compute, Technical University of Denmark, e-mail: aeha@dtu.dk
[2] Department of Mathematics and Computer Science, University of Bremen, e-mail: jp@cs.uni-bremen.de

Abstract. In this article, the feasibility of a unified modelling and programming paradigm is discussed from the perspective of large scale system development and verification in collaborative development environments. We motivate the necessity to utilise multiple formalisms for development and verification, in particular for complex cyber-physical systems or systems of systems. Though modelling, programming, and verification will certainly become more closely integrated in the future, we do not expect a single formalism to become universally applicable and accepted by the development and verification communities. The multi-formalism approach requires to translate verification artefacts (assertions, test cases, etc.) between different representations, in order to allow for the verification of emergent properties based on local verification results established with different methods and modelling techniques. It is illustrated by means of a case study from the railway domain, how this can be achieved, using concepts from the theory of institutions. This also enables the utilisation of verification tools in different formalisms, despite the fact that these tools are usually developed for one specific formal method.

1 Introduction

State of practise. "Programs are models" - this is a well known slogan of model-driven development, and it is well-founded, since several modelling formalisms allow for automated code generation, where the code is just regarded as a less abstract (textual) model. Refinement relations between abstract and more concrete models can be specified with mathematical rigour, so that the code can be traced back to the original model without any ambiguities. This suggests a unified approach to constructing models and software code during the development life cycle, and it also indicates that a comprehensive approach to developing new modelling formalisms as well as new programming languages should be adopted.

Both ideas, however, cannot be considered as state of practise today. Despite all efforts to provide a seamless modelling and programming environment, tool support for high-level modelling is still very heterogeneous and controversially discussed, with issues such as

– SCADE or Simulink/Stateflow or UML?

– Domain-specific languages or wide-spectrum languages?
– Semi-formal of fully formal modelling semantics?

In contrast to this, we experience a growing consensus about the effectiveness of programming languages (C++, Java, Haskell) and the usability of integrated development environments (IDEs), such as Eclipse, Microsoft Developer, or X-code, all of them offering effective support for software development, debugging, and various aspects of testing. Last, but not least, powerful re-usable libraries have been built for supporting efficient software programming, while the higher-level formalisms only provide very basic packages that can be re-used when creating models[3].

Summarising, we agree with the authors of [7] that it will take another 10 years until graphical high-level modelling tools will have reached a level of perfection that is comparable to current state-of-the art IDEs. According to our understanding, the reluctant acceptance of high-level modelling techniques is less caused by a reluctance to adopt formal concepts, but simply by the fact that the user experience is more satisfactory and the feeling of productiveness is higher when working on the level of software code than when working with more abstract formalisms.

Advocating the multi-formalism approach. This position paper is less about the closer integration of modelling, programming, and verification, because this is not the only problem to be solved: from our perspective, an even more severe problem consists in the fact that as of today, there is no preferred modelling formalism the majority of the development and verification communities might be willing to agree upon. On the contrary, development and verification projects for large and complex systems involving heterogeneous components, such as cyber-physical systems or systems of systems [12] suggest that a multi-formalism approach – supported by collaborative distributed development environments – may become the preferred solution in the future. This enables development and verification teams to use optimised methods and tools for developing and verifying specific system components. While this obviously helps to avoid endless discussions about which modelling language to choose in a project, the multi-formalism approach also comes with a down-side: verification results locally obtained for system components by means of different formalisms need to be translated into other representations when emergent system properties have to be derived from local component-specific assertions.

The main message of this paper is that the advantages of the multi-formalism approach outweigh this translation effort, because systematic methods for transferring theories and verification results between formalisms exist and can be efficiently applied. They even help to re-use tools built for one formalism in the context of another. We expect further that the necessity to translate verification artefacts between different formalisms will advance the integration of modelling

[3] It is interesting to note that the Z specification language already provided extensive libraries, as can be seen in its early reference books like [16]. This, however, has not become a standard requirement for designing new formalisms.

and programming, because these translations can only be defined and applied on the more abstract level of modelling.

Our thesis is supported further by the growing interest in model-driven systems engineering [13]: currently, manufacturers in the aircraft, railways, and automotive domains express considerable interest in a model-based approach to developing large-scale complex systems or even systems of systems. This interest is motivated by the desire to analyse executable models for early detection of conceptual errors and to exchange semantically precise models instead of or in addition to informal textual documents with suppliers. This general acceptance of the importance of formalised modelling is expected to accelerate the elaboration of integrated modelling and programming approaches. Moreover, at least for the avionic domain and for the domain of railway control systems, model-based systems engineering is always discussed in multi-formalism context, where tool-supported methods like Simulink/Stateflow, SCADE, SysML, and B are applied to different modelling, code generation, and verification tasks in large-scale development projects.

Overview. In Section 2, the necessity for a multi-formalism approach to large-scale system developments is justified. We analyse the possibilities to apply multiple high-level formalisms when developing and verifying complex systems in a collaborative development environment. In Section 3, a case study is presented that will be used to illustrate the verification of emergent properties in the next Section 4 using the linking approach which allows to translate assertions elaborated in one formalism to equivalent assertions of another. In Section 5, the linking approach is applied again: it is described how verification tools can interact to support different formalisms with a maximal degree of re-use. As an example, we consider test strategies for finite state machines with guaranteed fault coverage and show how the resulting test strategies can be translated to other formalisms while preserving the fault coverage properties. In Section 6 the conclusions are presented.

2 A Multi-formalism Approach to Large-scale System Developments

For large-scale system developments, as needed for complex distributed cyber-physical systems (CPS) or systems of systems, several modelling formalisms, and associated development and verification methods with corresponding tool support are needed. We see the following main reasons for this assessment.

- Sub-components should be modelled, developed, and verified with formalisms that are optimised for their specific requirements. For CPS, these components may be very heterogeneous, from smart sensors to discrete or hybrid (mixed discrete and time-continuous observables) control components, supported by database servers and mathematical constraint solvers.

– Different development and verification teams will work on large scale developments, each group preferring to use their "favourite" methods and associated tool box.

In a development campaign for an aircraft, for example, hybrid control tasks might be modelled with Simulink/Stateflow, local synchronous discrete control with SCADE, and integration aspects (such as asynchronous data exchange between flight deck and cabin) with SysML. This example shows that the multi-formalism approach is not so much a vision but more like an established fact today. What is missing is a systematic approach allowing for mathematically sound integration of development artefacts and for sound interpretation of local verification results in the global system context:

– For the verification of *emergent properties* – these are properties that can only be derived from the collaborative behaviour of all interacting system components – it is necessary to take local verification results into account which have been developed using different formalisms to express the assertions guaranteed by the sub-components.

Apart from this essential prerequisite for developing safe systems using the multi-formalism approach, there is another, more efficiency-oriented challenge to be solved:

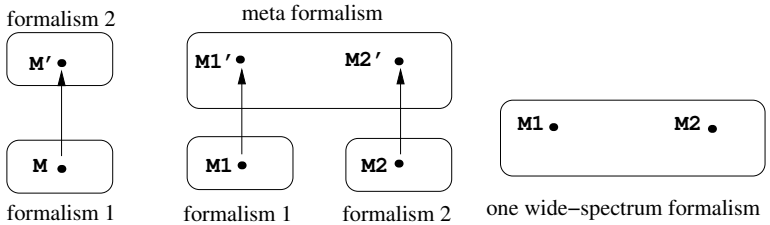
– Complex automation algorithms available in a tool supporting a specific formalism need to be made available for other formalisms as well.

Three approaches supporting multi-formalism development and verification. We see at least three possible major approaches for semantic integration of development and verification artefacts based on different formalisms:

1. The *linking approach* uses mechanisms to translate models and assertions between different formalisms. It can be based, for example, on the theory of institutions [5, 4, 6] with its foundations in mathematical category theory, or on the Unifying Theories of Programming (UTP) [8] which relies on lattice theory and the equivalence between programming languages and logic.
2. The *megamodel approach* maps the meta models (i.e. semantic models) of different formalisms into the same meta-meta model, usually denoted as a megamodel in the software engineering communities [1]. Megamodels contain the semantic representations of models elaborated in different formalisms; moreover, they contain transformations relating elements of the different models. This allows for verifying emergent properties on the megamodel representations of all sub-component assertions.
3. The *wide-spectrum approach* combines multiple "sub-formalisms" in one and provides a common "heavy weight" semantic meta model.

The three approaches are illustrated in Fig. 1.

From today's perspective, we consider the linking approach as the most promising one, because it has been elaborated in the most thorough way and



(a) Linking approach. (b) Megamodel approach. (c) Wide-spectrum approach.

Fig. 1. Three approaches for semantic integration.

exercised – sometimes only heuristically – in practise. The megamodel approach is still a fairly new research topic, and many investigations are only concerned with static model semantics, while the linking approach fully supports the translation of facts about behavioural semantics. Finally, the wide-spectrum approach (UML/SysML are prominent examples from this class) never seems to cover all modelling features that people may need in a specific development and verification undertaking. Moreover, the integration of sub-formalisms automatically results in complex semantic models. This can be seen in the UML and SysML standards published by the Object Management Group, where only the static semantics is fully formalised, while the behavioural semantics of language elements is only specified in natural language style.

The linking approach is illustrated by means of an example in the paragraphs below. We use linking techniques based on the theory of institutions. This should not suggest, however, that we consider this as the preferred linking method; the UTP-based method seems to be equally well-suited, as can be seen from case studies like [2].

3 Case Study – On-board Train Controller for Speed and Brakes

Consider an on-board control system for high-speed trains. This typically comprises several controllers communicating over some local bus system. In Fig. 2, a vital part of the on-board control system is shown, consisting of the *ceiling speed monitor (CSM)* and the *brake controller (BC)*. The former compares the train's current speed v against the maximal speed v_m currently admissible according to the commands received from the radio block centre. If the speed is too high, the CSM first sends warning messages to the train engine driver (N = Normal, O = Overspeed, W = Warning), and then – if the train is speeding even more – transits to intervention mode (I), where a braking command $b := \text{ON}$ is transmitted on the bus. The conditions to release the brakes after an intervention by the CSM depend on the train's location; this is reflected by a Boolean parameter *nva* ("national value allowing early release of brakes") sent by the radio block

centre. Its meaning is discussed below when presenting the behavioural CSM model. We assume that the CSM interface is realised according to the shared variable paradigm: the inputs are polled regularly, and the output variables are written to at the end of each processing cycle.

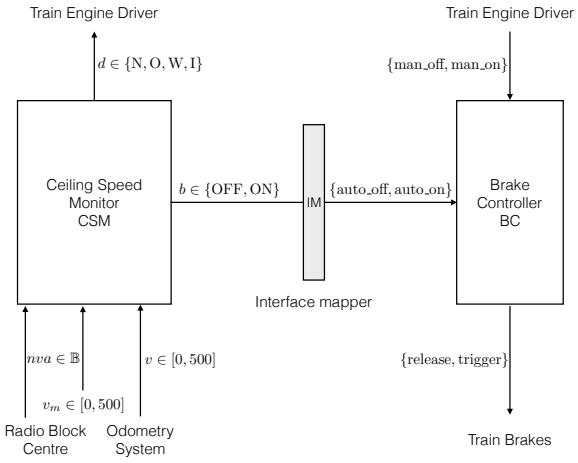


Fig. 2. Interfaces of the on-board train control system.

For illustrating certain aspects of theory linking, we assume that the BC has an event-based interface: it receives commands to trigger or release the brakes from both the CSM and the train engine driver as events auto_on, auto_off (automated trigger and release of the brakes) and man_on, man_off (manual trigger and release events), respectively. To map the state-based CSM output b to input events of the BC, an interface mapper (IM) observes changes of b and creates the corresponding auto_on, auto_off events for the BC. The IM could be implemented, for example, as a lower software layer of the BC which reads state data from the communication bus (realised, for example, as a reflective memory), and creates events for the BC software accordingly.

The CSM behaviour is specified by means of a SysML state machine communicating via shared variables, as shown in Fig. 3. Initially, the controller is in state NORMAL, and the outputs to train engine driver and BC are N and OFF, respectively. As soon as the actual speed exceeds the maximal speed allowed ($v > v_m$), the controller changes into state OVERSPEED and changes the indication to the train engine driver to O. If the actual speed exceeds $v_m + dW(v_m)$ (dW is a continuous non-negative function depending on v_m), the indication changes to W. Speeding further until the threshold $v_m + dI1(v_m)$ is violated leads to a transition into the first intervention state: the indication changes to I, and the output b to the BC is set to ON.

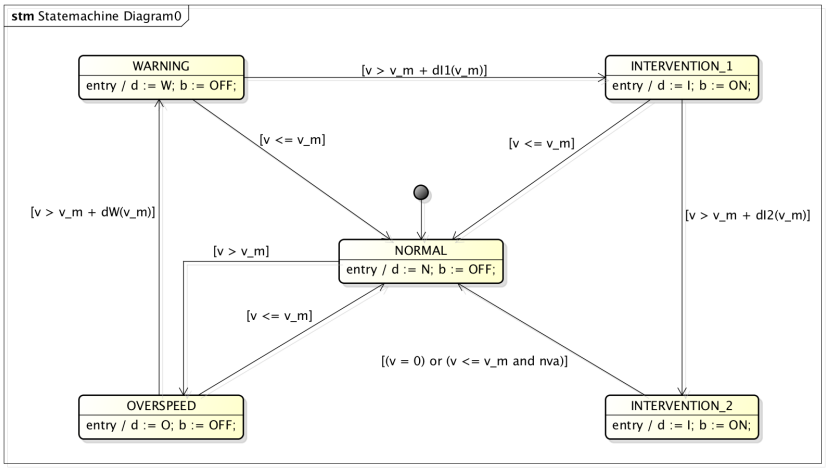


Fig. 3. SysML state machine model for the ceiling speed monitor.

In the control states described so far, the controller transits back to NORMAL and resets both indication and braking command, as soon as the actual speed is in normal range $v \leq v_m$ again. Further acceleration until $v > v_m + dI2(v_m)$, however, enforces a transition into state INTERVENTION_2. There the indication $d = \mathrm{I}$ and the output $b = \mathrm{ON}$ remain the same as in INTERVENTION_1, but the transition back to normal is only allowed when the train has come to a standstill, or if the country-dependent value nva has been set to `true` and $v \leq v_m$ holds again.

The brake controller BC is modelled as a deterministic finite state machine (DFSM) in Mealy Machine style, as shown in Fig. 4. In initial state RELEASED, the brakes are released, and repeated auto_off, man_off events do not change this. On reception of the man_on event from the train engine driver, the brakes are triggered. In the corresponding DFSM state TRIGGERED, the brakes may only be released again by the train engine driver: auto_off commands from the CSM are ignored. If however, the CSM also sends a braking command via event auto_on, the DFSM transits into state TRIGGERED_AUTO, and now only the auto_off command can release the brakes again. If the BC is in state RELEASED and gets the command to trigger the brakes from the CSM, it directly transits to TRIGGERED_AUTO. Again, only the CSM command can initiate the release of the brakes in this situation, and commands from the train engine driver are ignored.

4 Emergent Property Verification

In the case study introduced above, consider the safety-related verification obligation

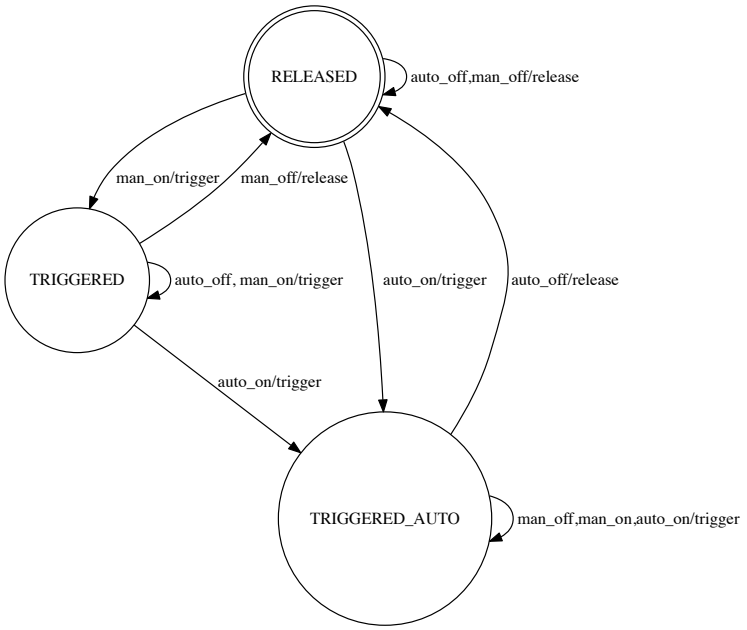


Fig. 4. Finite state machine model of the brake controller.

> *Whenever the CSM indicates intervention (I) on output d, the emergency brakes are triggered in the next system state at the latest.*

This is an emerging property, because its validity cannot be decided by analysis of the CSM or the BC alone: the CSM knows about d, but it has no control over the trigger event to the brakes. On the contrary, the BC knows when the trigger event has been fired, but doesn't know about d-indications.

Furthermore, we observe that the CSM and the BC have been modelled with different formalisms, since the latter has DFSM semantics, while the former is represented as a SysML state machine with a shared variable interface. The behavioural semantics of SysML state machines can be represented conveniently by Kripke Structures (see [10] for a detailed description of the CSM semantics) whose states are variable valuation functions and whose atomic propositions have CSM variables and control state names as free variables. We choose the semantic variant where outputs changed while passing through transient states are not observable. Assume, for example, that the CSM is in state NORMAL and receives a new actual speed value $v > v_m + dI2(v_m)$. Then it passes through states OVERSPEED, WARNING, and INTERVENTION_1 until it ends up in INTERVENTION_2 where it becomes stable. Only then the associated outputs $d = \mathrm{I}, b = \mathrm{ON}$ become visible. Moreover, input changes only become visible while the CSM resides in a stable state.

We decide to use Kripke Structures to represent the semantics of the complete system $\mathcal{S}$ and specify assertions by means of LTL formulas over variable symbols of $\mathcal{S}$. The variables of $\mathcal{S}$ are the interface variables of the CSM plus auxiliary Boolean variables a_{on} (if true, the last input event on the IM-BC interface was auto_on, if false, the last input on this interface was auto_off), m_{on} (if true, the last input event to the BC on the train engine driver interface was man_on, if false, it was man_off), r (if true, the last output event of the BC was release, if false, the last output was trigger).

With these variable symbols at hand, the safety property specified textually above can be formalised as

$$\varphi_{\mathcal{S}} \equiv \mathbf{G}\big(\neg(d = \text{I}) \vee \mathbf{X}\neg r\big) \tag{1}$$

Let us now assume that local verification activities have already shown that the CSM implementation conforms to the SysML model shown in Fig. 3, the BC implementation conforms to the model in Fig. 4, and that the interface mapper implementation fulfils

$$\varphi_{\text{IM}} \equiv \mathbf{G}\big((b = \text{ON}) \Leftrightarrow \mathbf{X}a_{\text{on}}\big) \tag{2}$$

Formula (2) specifies that the IM reacts on a change of CSM output b from OFF to ON by creating an auto_on event which becomes visible to the BC in the next processing step. Conversely, if the CSM changes the b output from ON to OFF, the IM generates an auto_off event which is reflected by $\neg a_{\text{on}}$ holding in the next processing step.

Using k-induction [14], we can prove by model checking that the CSM model satisfies the invariant

$$\varphi_{\text{CSM}} \equiv \mathbf{G}\big(\neg(d = \text{I}) \vee (b = \text{ON})\big) \tag{3}$$

when interpreted in the semantics where intermediate transient processing steps are not observable.

It remains to establish a suitable LTL assertion for the BC. This is not straightforward, since the DFSM semantics is represented by the state machine's *language* $L(BC)$, consisting of all finite traces of input/output events that can be performed by the BC. In contrast to this, LTL formulas are interpreted over infinite sequences π of sets of valid atomic propositions. We observe, however, that the violation of every *LTL safety property* can already be decided on a finite prefix of π. Conversely, every finite execution prefix not violating a safety formula can be extended to an infinite execution π which is a model for the safety formula [15]. Since all LTL formulas which are of interest in our context are safety formulas, this suggests that the DFSM language $L(BC)$ can be interpreted to fulfil a safety property, if this property is not violated by any I/O trace of the language.

This concept will now be realised formally by mapping DFSMs with the signature of the BC to associated Kripke structures with atomic propositions $a_{\text{on}}, m_{\text{on}}, r$. The construction follows the recipes of the theory of institutions [5,

4, 6]. An institution defines some essential aspects of a logic system: signatures, sentences (over a given signature), models (over a given signature), and the satisfaction relation between models and sentences of the same signature. Mappings between two institutions can be defined, by defining maps translating signatures, sentences and models between the two institutions, respectively. The linking approach performed in this section, uses this idea to first define a model map μ translating DFSM models having the signature Σ_{BC}[4] of the BC DFSM to Kripke models over a corresponding Kripke signature Σ_K[5]. We can then use this model translation map to translate the DFSM model of the BC to a corresponding Kripke model for which we can formulate a suitable LTL assertion. To ensure that the behaviour of the translated model is consistent with the original model, we also define a sentence translation map σ from Kripke sentences (LTL assertions) over Σ_K to DFSM sentences over Σ_{BC} and prove a satisfaction condition that expresses that model satisfaction of sentences is "invariant under change of formalism". The translation of models and sentences is illustrated in Fig. 5.

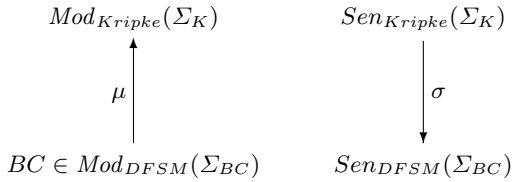


Fig. 5. Model and sentence translation. $Mod_{Kripke}(\Sigma_K)$ is the set of all Kripke models over Σ_K, $Mod_{DFSM}(\Sigma_{BC})$ is the set of all DFSM models over Σ_{BC}, $Sen_{Kripke}(\Sigma_K)$ is the set of all Kripke sentences over Σ_{BC}, and $Sen_{DFSM}(\Sigma_{BC})$ is the set of all DFSM sentences over Σ_{BC}.

Model map. As a first step, the *model map* μ mapping DFSMs over signature Σ_{BC} to Kripke Structures over signature Σ_K is created. Let $M = (Q, q_0, \Sigma_I, \Sigma_O, h)$ be a DFSM with finite state space Q, initial state $q_0 \in Q$, input alphabet $\Sigma_I = \{\text{auto\_on}, \text{auto\_off}, \text{man\_on}, \text{man\_off}\}$, output alphabet $\Sigma_O = \{\text{release}, \text{trigger}\}$, and transition relation $h \subseteq Q \times \Sigma_I \times \Sigma_O \times Q$. Then $\mu(M)$ is defined as the Kripke Structure $\mu(M) = (S, s_0, R, L, AP)$ with

1. Atomic proposition set $AP = \{a_{\text{on}}, m_{\text{on}}, r\}$
2. State space $S \subseteq Q \times 2^{AP}$
3. Initial state $s_0 = (q_0, \{r\})$
4. Labelling function $L : S \to 2^{AP}; \quad (q, A) \mapsto A$
5. Transition relation $R \subseteq S \times S$ specified by

[4] The signature of a DFSM model consists of its input alphabet and output alphabet. For the BC model, we have $\Sigma_{BC} = (\Sigma_I, \Sigma_O) = (\{\text{auto\_on}, \text{auto\_off}, \text{man\_on}, \text{man\_off}\}, \{\text{release}, \text{trigger}\})$.

[5] A Kripke signature consists of those input variables, local variables and output variables that can be used in a model over that signature. For the corresponding Kripke signature Σ_K it is the variables a_{on}, m_{on}, and r.

$$
\begin{aligned}
R = \{((q,A),(q',A')) \in S \times S \mid (q, \text{man\_on}, \text{release}, q') \in h \wedge A' = A \cup \{m_{\text{on}}, r\}\} \\
\cup \{((q,A),(q',A')) \in S \times S \mid (q, \text{man\_on}, \text{trigger}, q') \in h \wedge A' = (A \setminus \{r\}) \cup \{m_{\text{on}}\}\} \\
\cup \{((q,A),(q',A')) \in S \times S \mid (q, \text{man\_off}, \text{release}, q') \in h \wedge A' = (A \setminus \{m_{\text{on}}\}) \cup \{r\}\} \\
\cup \{((q,A),(q',A')) \in S \times S \mid (q, \text{man\_off}, \text{trigger}, q') \in h \wedge A' = (A \setminus \{m_{\text{on}}, r\})\} \\
\cup \{((q,A),(q',A')) \in S \times S \mid (q, \text{auto\_on}, \text{release}, q') \in h \wedge A' = A \cup \{a_{\text{on}}, r\}\} \\
\cup \{((q,A),(q',A')) \in S \times S \mid (q, \text{auto\_on}, \text{trigger}, q') \in h \wedge A' = (A \setminus \{r\}) \cup \{a_{\text{on}}\}\} \\
\cup \{((q,A),(q',A')) \in S \times S \mid (q, \text{auto\_off}, \text{release}, q') \in h \wedge A' = (A \setminus \{a_{\text{on}}\}) \cup \{r\}\} \\
\cup \{((q,A),(q',A')) \in S \times S \mid (q, \text{auto\_off}, \text{trigger}, q') \in h \wedge A' = (A \setminus \{a_{\text{on}}, r\})\}
\end{aligned}
$$

The Kripke Structure created via μ from the BC DFSM is shown in Fig. 6. The initial state is (RELEASED, $\{r\}$). Its reachable states are the pairs (q, A) of BC states and subsets $A \subseteq AP$, such that the latter are always consistent with the latest input events: after an event man_on has occurred on DFSM level, the associated target state in $\mu(BC)$ contains atomic proposition m_{on}. If this is followed by DFSM input auto_on, then a_{on} is added to the propositions of the $\mu(BC)$ target state. On a path of state transitions in $\mu(BC)$, m_{on} remains in the set of atomic propositions associated with each state until a state transition corresponds to a DFSM transition triggered by input man_off, whereupon m_{on} is removed from the propositions of the target state reached by $\mu(BC)$.

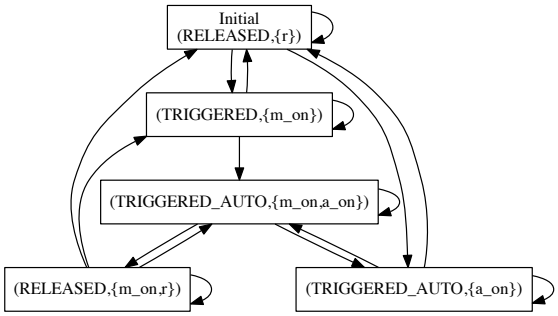


Fig. 6. Kripke Structure $\mu(BC)$.

Sentence translation map. In the second step of the linking approach, a *sentence translation map* σ is created. This is a map allowing us to translate assertions defined for Kripke Structures into assertions about DFSMs. For the purpose of this small example we can restrict the sentences of interest to LTL invariants $\mathbf{G}\psi$, where ψ is a proposition in negation normal form, built from the atomic propositions of $AP = \{a_{\text{on}}, m_{\text{on}}, r\}$.

On the DFSM level, sentences are predicates over I/O-traces ι, implicitly quantified over all $\iota \in L(M)$, where $L(M)$ is the set of all I/O-traces of the

model M under consideration. Again, we restrict these predicates to invariants that are written in LTL style; more precisely:

1. Sentences over a signature (Σ_I, Σ_O) are of the form $\mathbf{G}\alpha$, where α is a predicate in negation normal form using atomic propositions from the set

$$AP_M = \{x = c \mid c \in \Sigma_I\} \cup \{y = e \mid e \in \Sigma_O\}$$

2. The satisfaction relation between models M and sentences $\mathbf{G}\alpha$ consists of invariant assertions over $L(M)$ for DFSMs models M written in the form $M \models \mathbf{G}\alpha$ which is interpreted as

$$M \models \mathbf{G}\alpha \equiv \forall \iota \in L(M) : \mathbf{G}\alpha(\iota)$$

where $\mathbf{G}\alpha(\iota)$ is interpreted as

$$\mathbf{G}\alpha(\iota) \equiv \forall i = 1, \ldots, n : \alpha[x_i/x, y_i/y]$$

for $\iota = (x_1, y_1).(x_2, y_2) \ldots (x_n, y_n)$. In this definition, $\alpha[x_i/x, y_i/y]$ denotes the proposition α with every occurrence of x replaced by the actual input event x_i, and every y replaced by the actual output y_i.

Take, for example, the assertion

$$BC \models \mathbf{G}(\neg(x = \text{man\_off}) \vee (y = \text{release}))$$

This assertion is not fulfilled, because the BC can perform the I/O-trace

$$(x_1, y_1).(x_2, y_2).(x_3, y_3) \cdots = (\text{man\_on}, \text{trigger}).(\text{auto\_on}, \text{trigger}).(\text{man\_off}, \text{trigger}) \ldots$$

Evaluating $\alpha[x_3/x, y_3/y]$ results in

$$\alpha[x_3/x, y_3/y] \equiv \neg(\text{man\_off} = \text{man\_off}) \vee (\text{trigger} = \text{release}) \equiv \texttt{false}$$

Let W_K denote the invariant LTL formulas $\mathbf{G}\psi$ over Kripke Structures, and W_M the invariant formulas $\mathbf{G}\alpha$ over I/O-sequences of DFSMs. Then the sentence translation map can be defined as follows.

$$\begin{aligned}
&\sigma : W_K \longrightarrow W_M; \qquad \mathbf{G}\psi \mapsto \mathbf{G}(\sigma'(\psi)) \\
&\sigma' : \text{Propositions}(AP) \longrightarrow \text{Propositions}(AP_M) \\
&\quad m_{\text{on}} \mapsto (x = \text{man_on}) \\
&\quad \neg m_{\text{on}} \mapsto (x = \text{man_off}) \\
&\quad a_{\text{on}} \mapsto (x = \text{auto_on}) \\
&\quad \neg a_{\text{on}} \mapsto (x = \text{auto_off}) \\
&\quad r \mapsto (y = \text{release}) \\
&\quad \neg r \mapsto (y = \text{trigger}) \\
&\quad \psi \wedge \psi' \mapsto \sigma'(\psi) \wedge \sigma'(\psi') \\
&\quad \psi \vee \psi' \mapsto \sigma'(\psi) \vee \sigma'(\psi')
\end{aligned}$$

Satisfaction condition. Having constructed model map and sentence translation map, the so-called *satisfaction condition* has to be proven. The satisfaction condition states in our case that

$$\mu(M) \models \mathbf{G}\psi \text{ if and only if } M \models \sigma(\mathbf{G}\psi)$$

It is straightforward to see that this follows directly from the way μ and σ have been constructed. The satisfaction condition is illustrated in Fig. 7.

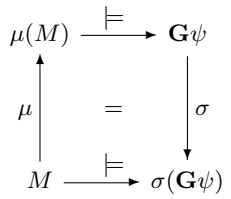


Fig. 7. Satisfaction condition for model and sentence translation.

Other conditions. It also to be proven that the model map μ is properly defined in the sense that it preserves the "natural" morphisms between models. For DFSMs, these morphisms are arrows indicating I/O-equivalence: $M_1 \longrightarrow M_2$ if and only if $L(M_1) = L(M_2)$, and therefore also an arrow $M_2 \longrightarrow M_1$ exists. On the level of Kripke Structures, the corresponding morphisms are bisimulations between Kripke Structures defined over the same atomic propositions. It is easy to see that μ maps I/O-equivalent DFSMs to bisimilar Kripke Structures, so this condition is fulfilled. The condition is illustrated in Fig. 8.

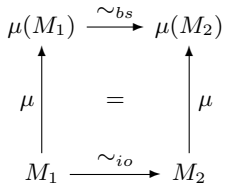


Fig. 8. The model map μ translates I/O-equivalent DFSMs to bisimilar Kripke Structures. $\sim_{io}$ denotes the io-equivalence relation and $\sim_{bs}$ denotes the bisimulation relation.

Proof of emergent property. Having established the satisfaction condition, we are now in the position to represent properties of the BC by means of LTL invariants over atomic propositions from AP, and every invariant that can be shown

for $\mu(BC)$ is ensured by the BC itself, just in the slightly differing syntactic representation of W_M-formulas.

Analysing the transition graph of $\mu(BC)$ in Fig. 6, it can be immediately deduced that the invariant

$$\varphi_{\mathrm{BC}} \equiv \mathbf{G}\big(\neg a_{\mathrm{on}} \vee \neg r\big) \tag{4}$$

is fulfilled. Collecting now the assertions established in (2), (3), and (4), it is easy to see that together they imply the desired safety property specified in (1). □

The example above illustrated the application of the theory of institutions in a linking approach to verify emergent properties in a large scale system development, where different formalisms are used for modelling different system components. In the next section, another application of this approach will be described: the cooperation of tools, each of them fulfilling verification tasks for specific formalisms.

5 Collaborative Development and Verification Environments – Next Generation

Large scale system developments require *collaborative development environments (CDEs)*, where geographically distributed development teams can work locally on their specific components and cooperate on integration tasks. We expect that the CDE paradigm will become even more popular in the future, because it will also enable collaboration of tools, with the objective to support the multi-formalism approach. This might be particularly beneficial for verification tools.

Complete DFSM test strategies. To illustrate this point, let us consider a model-based test automation tool available for DFSMs, that applies so-called *complete test strategies*: this means that a test suite generated from a reference model M

1. accepts every implementation M' fulfilling the given conformance relation $M' \leq M$ (soundness), and
2. rejects every implementation M' violating $M' \leq M$ by letting at least one test case fail (exhaustiveness).

For black-box testing, completeness is always defined in relation to a fault domain. For DFSMs with I/O-equivalence as conformance relation, for example, complete test strategies usually depend on the fault domain $\mathcal{D}(\Sigma_I, \Sigma_O, m)$ containing all DFSMs over signature (Σ_I, Σ_O), whose minimised equivalent DFSM contains at most m states. For this fault domain, several practically implementable test strategies exist [3, 17, 11].

Complete test strategies for the CSM. Suppose that the model-based DFSM testing tool was available and could be applied for testing the brake controller BC. The superior test strength of complete test suites suggests to investigate

whether such a strategy might also be available for the ceiling speed monitor CSM. This requires some consideration, since the CSM has inputs v, v_m which are of floating point type. Therefore it is infeasible to enumerate all possible input combinations during a test suite, so we cannot simply represent the CSM as another DFSM. It is possible, however, to construct input equivalence classes for the SysML state machine, because only the CSM inputs cannot be enumerated, but the internal states and its outputs are finite. The construction of these classes has been elaborated in [10], and it has been shown that this enables an abstraction of the SysML state machine to a DFSM with a signature $\Sigma_{CSM'}$ having input equivalence classes as inputs and with an output alphabet corresponding to the finite value assignments to the CSM outputs d and b. In the light of the linking approach, this result can be re-phrased as follows and as illustrated in Fig. 9.

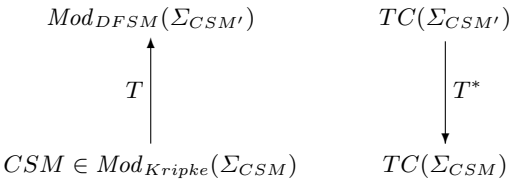


Fig. 9. Model translation T and test case translation T^*. Σ_{CSM} is the signature of the CSM model (including the input variables nva, v, and v_m, and the output variables b and d) and $\Sigma_{CSM'}$ is the corresponding DFSM signature. $Mod_{Kripke}(\Sigma_{CSM})$ is the set of all Kripke models over the signature Σ_{CSM}, $Mod_{DFSM}(\Sigma_{CSM'})$ is the set of all DFSM models over $\Sigma_{CSM'}$, $TC_{Kripke}(\Sigma_{CSM})$ is the set of all Kripke test cases over Σ_{CSM}, and $TC_{DFSM}(\Sigma_{CSM'})$ is the set of all DFSM test cases over $\Sigma_{CSM'}$.

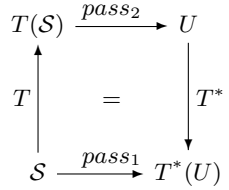


Fig. 10. Satisfaction condition for model and test case translation.

1. Every deterministic Kripke Structure $\mathcal{S}$ with infinite input domains but finite internal state values and finite outputs can be mapped by the model translation map T to a minimised DFSM. This DFSM take input equivalence classes of $\mathcal{S}$ as inputs and operates on the same outputs as $\mathcal{S}$.
2. The model translation map T respects I/O-equivalence as conformance relation: if Kripke Structure $\mathcal{S}'$ conforms to the reference model $\mathcal{S}$, then the DFSM $T(\mathcal{S}')$ conforms to the reference DFSM $T(\mathcal{S})$, as illustrated in Fig. 11.

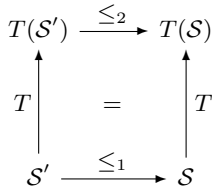


Fig. 11. Model translation preserves conformance relation.

3. Sentences in this scenario are test cases and the sentence translation map is the *test case map* T^* which translates DFSM test cases into test cases running against implementations with Kripke Structure semantics.
4. Satisfaction relations are now of the form "FSM M passes a test case" or "Implementation $\mathcal{S}$ passes a test case".
5. The diagram in Fig. 10 commutes; this implies that (T, T^*) fulfil the satisfaction condition: the DFSM abstraction $T(\mathcal{S})$ of implementation $\mathcal{S}$ passes a DFSM test case U, if and only if the implementation $\mathcal{S}$ also passes the translated test case $T^*(U)$ on Kripke Structures.
6. The satisfaction condition now implies that complete test strategies derived from reference DFSM $T(\mathcal{S})$ are translated via T^* to likewise complete test strategies for implementations with Kripke Structure semantics. This shows that not only assertions about specific models, but also whole *theories* can be transferred from one institution to the other.

As a result of these theoretical considerations (they have been elaborated in more detail in [9]), we can construct a tool for testing implementations against SysML state machine models similar to the CSM model as follows.

1. Input reference model $\mathcal{S}$.
2. Calculate DFSM abstraction $T(\mathcal{S})$.
3. Send DFSM $T(\mathcal{S})$ to the DFSM testing tool to calculate the associated complete DFSM test suite $\mathbf{TS}(T(\mathcal{S}))$.
4. Receive $\mathbf{TS}(T(\mathcal{S}))$ from the DFSM testing tool and translate it to a complete test suite $T^*(\mathbf{TS}(T(\mathcal{S})))$ that can be executed against implementation $\mathcal{S}'$.

In [2], a similar approach to re-using tools in different formalisms is described; this is based on the Unifying Theories of Programming UTP. The role of the morphisms and co-morphisms between institutions is taken on by Galois connections between lattices in UTP.

6 Conclusions

In this contribution, the aspect of using multiple formalisms in large-scale system developments with collaborative development environments has been discussed.

For the domain of cyber-physical systems, we consider the multi-formalism approach as essential for such undertakings, because special methods and associated modelling techniques are needed to optimise the development and verification of system components possessing a considerable structural and behavioural variety – from smart sensors via mechatronics controllers to database servers. We have argued that the multi-formalism approach requires some theoretical support for verifying emergent system properties and re-using verification tools in the context of different formalisms. This has been illustrated by means of a case study for an on-board train control system, where the theory of institutions has been applied as one possibility for showing how assertions can be translated between different semantic domains and how test suites with guaranteed fault detection properties can be translated from one domain into another.

These considerations suggest that there is no single "best" unified modelling, programming and verification paradigm to be expected in the future. Instead, system development and verification according to the multi-formalism approach will become more and more natural. The examples show that this trend will automatically foster the integration between modelling and programming: the transfer of verification artefacts between different formalisms requires a level of abstraction which is significantly higher than that of typical programming languages.

Nevertheless, considerable work is still necessary to achieve a degree of usability and integration for modelling and verification techniques that is already available today for "conventional" programming in integrated development environments. In particular, it cannot be expected that the institution morphisms and co-morphisms together with their satisfaction conditions will be elaborated manually from scratch for every new large scale system development. Instead, a library of existing inter-formalism transformations is needed, and the (usually routine) proofs of satisfaction conditions need to be mechanised.

Acknowledgements. The first author's research has been funded by the RobustRailS project granted by Innovation Fund Denmark. The second author's contribution has been elaborated within project *ITTCPS – Implementable Testing Theory for Cyber-physical Systems* (http://www.cs.uni-bremen.de/agbs/-projects/ittcps/index.html) which has been granted by the University of Bremen in the context of the German Universities Excellence Initiative (http://en.wikipedia.org/wiki/German_Universities_Excellence_Initiative).

Some diagrams in this paper were created using Paul Taylors diagrams package.

References

1. J. Bézivin, F. Jouault, and P. Valduriez. On the need for megamodels. In *OOPSLA/GPCE: Best Practices for Model-Driven Software Development workshop*, 2004.
2. Ana Cavalcanti, Wen-ling Huang, Jan Peleska, and Jim Woodcock. CSP and kripke structures. In Martin Leucker, Camilo Rueda, and Frank D. Valencia, editors,

Theoretical Aspects of Computing - ICTAC 2015 - 12th International Colloquium Cali, Colombia, October 29-31, 2015, Proceedings, volume 9399 of *Lecture Notes in Computer Science*, pages 505–523. Springer, 2015.
3. Tsun S. Chow. Testing software design modeled by finite-state machines. *IEEE Transactions on Software Engineering*, SE-4(3):178–186, March 1978.
4. Razvan Diaconescu. *Institution-independent Model Theory*. Birkhäuser Verlag AG, Basel, Boston, Berlin, 2008.
5. J. A. Goguen and R. M. Burstall. Institutions: Abstract model theory for specification and programming. *Journal of the Association for Computing Machinery*, 39:95–146, 1992. Predecessor in: LNCS 164, 221–256, 1984.
6. Joseph Goguen and Grigore Roşu. Institution morphisms. *Formal Aspects of Computing*, 13(3):274–307, 2014.
7. Hans Grönniger, Holger Krahn, Bernhard Rumpe, Martin Schindler, and Steven Völkel. Textbased modeling. *CoRR*, abs/1409.6623, 2014.
8. C. A. R. Hoare and H. Jifeng. *Unifying Theories of Programming*. Prentice-Hall, 1998.
9. Wen-ling Huang and Jan Peleska. Complete model-based equivalence class testing for nondeterministic systems. *Formal Aspects of Computing*. Under review.
10. Wen-ling Huang and Jan Peleska. Complete model-based equivalence class testing. *STTT*, 18(3):265–283, 2016.
11. G. Luo, G.V. Bochmann, and A. Petrenko. Test selection based on communicating nondeterministic finite-state machines using a generalized wp-method. *IEEE Transactions on Software Engineering*, 20(2):149–162, 1994.
12. Claus Ballegaard Nielsen, Peter Gorm Larsen, John Fitzgerald, Jim Woodcock, and Jan Peleska. Systems of systems engineering: Basic concepts, model-based techniques, and research directions. *ACM Comput. Surv.*, 48(2):18:1–18:41, September 2015.
13. Douglas C. Schmidt. Model-driven engineering. *IEEE Computer*, 39(2), February 2006.
14. Mary Sheeran, Satnam Singh, and Gunnar Stålmarck. Checking safety properties using induction and a sat-solver. In Warren A. Hunt Jr. and Steven D. Johnson, editors, *Formal Methods in Computer-Aided Design, Third International Conference, FMCAD 2000, Austin, Texas, USA, November 1-3, 2000, Proceedings*, volume 1954 of *Lecture Notes in Computer Science*, pages 108–125. Springer, 2000.
15. A. Prasad Sistla. Safety, liveness and fairness in temporal logic. *Formal Aspects of Computing*, 6(5):495–511, September 1994.
16. J. M. Spivey. *The Z Notation: A Reference Manual*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1989.
17. M. P. Vasilevskii. Failure diagnosis of automata. *Kibernetika (Transl.)*, 4:98–108, July-August 1973.